INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

3A STI

Compilation

Cahier de TD/TP, printemps 2020

C. Alias

TD1 : Traduction dirigée par la syntaxe

Dans ce premier TD, on va construire un compilateur pour un sous-ensemble très simple du langage C (pas de fonctions, uniquement des variables entières) vers une machine à pile. Dans les TD suivants, on verra comment enrichir ce compilateur (fonctions, données structurées) et comment améliorer la qualité du code machine produit.

1 Machine à pile

Voici deux programmes en code machine. On nomme x la variable stockée au slot 0 ($\delta(x) = 0$) et y la variable stockée au slot 1 ($\delta(y) = 1$).

```
alloc 2
push 0
push 1
add
mpush 1
sub
pop 0
free
```

```
alloc 2
push 0
push 1
testg
jz end
push 0
push 1
sub
pop 0
end:
free
```

Pour chaque programme : que fait-il ? Comment l'écririez vous en `C` ?

2 Traduction dirigée par la syntaxe

On considère le programme suivant :

```
int x,y;
while(x != y)
  if(x>y) x = x - y;
  else y = y - x;
```

1. Proposez une règle de traduction pour le while.
2. Traduire le programme en déroulant les régles de traduction.

3 Implémentation

On va maintenant implémenter les régles de traduction. Décompacter le fichier `scc-light.tgz`. Notre compilateur comporte les fichiers suivants :

Fichier	Contenu
parser.ypp	Grammaire attribuée : **fichier à compléter**
lexer.l	Analyseur lexical
SymbolTable.*	δ : variable $\mapsto$ slot de pile
Backend-stack.*	Production de code pile
Backend-x86.*	Production de code x86

Inspecter `parser.ypp` et compléter le tableau suivant :

Non-terminal	Catégorie syntaxique	Exemple
expr	Expressions	`2*x + 1`
test		
stmt		
declare_local_vars		
prog		

Compilation des expressions

Compléter les règles de `expr` et compiler avec `make`. Le binaire du compilateur est `bin/scc`. Aller dans `test/`. Tester sur `expr.c` avec `../bin/scc expr.c`. Modifiez `expr.c` à votre guise.

Compilation des conditions

Même chose avec `test`. Tester sur `test.c`.

Compilation du contrôle

Même chose avec `stmt`. Tester sur `pgcd.c`.

Génération de code x86-64

On souhaite compiler vers x86-64. Une technique très simple (et très inefficace) est de traduire directement chaque instruction pile en instructions x86-64.

1. Inspectez les fichiers `Backend-stack.*` et `Backend-x86.*`
2. Quel backend est utilisé pour le moment ? Faire le swap en remplaçant `Backend-stack.o` par `Backend-x86.o` dans le `Makefile`. Tester sur `simple.c` avec `make simple.out` et sur `pgcd.c` avec `make pgcd.out`. Où se trouvent les implémentations des fonctions `input()` et `print()` ?

TD2 : Activations

Dans ce TD, on va étendre notre compilateur avec les fonctions. Plus précisément, on va compiler vers la machine à pile : une déclaration de fonction, un return et un appel de fonction.

Travail à rendre

Ce TD est noté. Vous rendrez une archive `tgz` avec :

– Un fichier `reponse.pdf` qui mentionne :

1. Votre nom, prénom, numéro d'étudiant. Le projet est individuel.
2. Vos réponses aux questions de la section **Implémentation**. Chaque réponse peut tenir en une phrase.
3. L'état de votre l'implémentation (quels exemples fonctionnent, quels exemples ne fonctionnent pas et pourquoi, comment compiler/tester votre compilateur). Quelques phrases suffisent.

– Votre implémentation du compilateur.

1 A la main

On considère les programmes suivants :

```
int f(int x)
{
  int result;
  result = x+1;
  //P
  return result;
}

void main()
{
  int result;
  result = f(1);
}
```

```
int pow(int x, int n)
{
  int result;
  if(n == 0)
    result = 1;
    //P
  else
    result = x*pow(x,n-1);
  return result;
}

void main()
{
  int result;
  result = pow(2,1);
}
```

1. Pour chaque programme, dessiner l'état de la pile au point d'exécution `//P`
2. Traduire la fonction `pow` en déroulant les régles de traduction vues en cours.

2 Implémentation

Décompacter le fichier scc.tgz. On retrouve la même organisation que scc-light.tgz :

Fichier	Contenu
parser.ypp	Grammaire attribuée : **fichier à compléter**
lexer.l	Analyseur lexical
SymbolTable.*	δ : argument/variable locale $\mapsto$ slot de pile
Backend-stack.*	Production de code pile
Backend-x86.*	Production de code x86

Inspecter `parser.ypp` et compléter le tableau suivant :

Non-terminal/règle	Catégorie syntaxique	Exemple
function	Déclaration de fonction	int fact(int n) { ... }
declare_args		
declare_local_vars		
règle l.123		
caller_arg_list		
règle l.200		
function_list		
prog		

Compilation des appels de fonction

On considère la règle l.123.

1. Comment les arguments sont poussés sur la pile ?
2. Ajouter le call (utiliser la fonction `call` du backend (voir `Backend.h`).
3. Où se trouve le résultat ?
4. Tester sur `call.c` avec `../bin/scc call.c`. Modifiez `call.c` à votre guise.

Compilation des fonctions

On considère la règle `function:`.

1. Quelle est l'action de declare_args et declare_local_vars ?
2. Compléter les parties manquantes.
3. Tester sur `call.c`

Compilation du `return`

On considère la règle du `return`.

1. Rappeler l'instruction à utiliser pour le return. Chercher dans `Backend.h` comment la produire.
2. Comment l'expression à retourner est elle poussée sur la pile ?
3. Compléter la règle.
4. Tester sur `call.c`

Validation du compilateur

On va produire du code x86-64 pour vérifier que les fonctions sont correctement compilées.

1. Dans le `Makefile`, substituer `Backend-stack.o` par `Backend-x86.o`.
2. Recompiler avec `make`.
3. Tester sur `call.c` (ajouter des `input()` et des `print()`), `fact.c`, `fib.c` et `pgcd.c`.

Compilation du `do/while`

La structure de contrôle `do S while(C)` exécute le bloc `S` tant que la condition `C` est vraie.

1. Proposer un schéma de traduction pour le `do/while`.
2. Implémenter le `do/while` dans le compilateur (on s'inspirera du `while`). Tester sur `dowhile.c`.

TD3 : Données structurées

Dans ce TD, on va voir comment compiler un programme C avec des données structurées (tableaux, structs, etc) vers une machine à registres. Plus précisément, comment compiler la *construction* d'une donnée structurée sur la pile, et comment compiler l'*accès* à une donnée structurée.

1 Machine à registres

Voici deux programmes en code machine. Le programme de gauche traduit une fonction `int f(int x, int y)`, celui de droite traduit une fonction `int g(int x, int y)` :

```
f: alloc ?
t1 = [$bp + 3]
t2 = [$bp + 2]
t3 = t1 + t2
t4 = 1
t5 = t3 - t4
t6 = t5
free
ret 2
```

```
g: alloc ?
t1 = [$bp + 3]
t2 = [$bp + 2]
t5 = 0
cjump t1>t2 --> then
jump endif
then:
push t1
t3 = t1 - t2
push t3
call f
t5 = t7
endif:
t4 = t5
free
ret 2
```

Pour chaque fonction :

1. Où se trouvent x et y ? Donner $\rho(x)$ et $\rho(y)$. Comment le résultat est-il retourné ?
2. Retrouver la fonction en C
3. Donner une réalisation possible des temporaires t1, t2, ... avec des registres x86-64 parmis $rax, $rbx, $rcx, $rdx, $r8, ..., $r15 ($bp est réalisé par $rbp) [a]
4. Dans quel cas a-t-on besoin de réaliser des temporaires avec des slots de pile ? Fixer le `?` de `alloc ?`.

a. Pour l'exercice, on garde les instructions de la machine à registre. Evidemment, il faudrait utiliser des instructions x86-64 ; on verra plus tard comment faire çà.

2 Organisation des données structurées

Dessiner l'organisation sur la pile des données suivantes :

```
struct{int a; int b;}[2] x;
```

```
struct {
  struct{int re; int im;} a;
  int[2] b;
} y;
```

3 Compilation des données structurées

On considère le programme suivant :

```
typedef struct {int x; int y; } point_t;
typedef struct {point_t A; point_t B; } segment_t;

void init(segment_t s, int n)
{
  s.A.x = n;
  s.A.y = n;
  s.B.x = n+1;
  s.B.y = n+1;
  //P
}

void main(int argc, (char*)* argv)
{
  segment_t s;
  init(s,argc);
}
```

1. Dessiner l'état de la pile ou point `//P`.
2. Donner le prélude et le postlude de `init`.
3. Que fait le code $[\![$ `s.B.y` $]\!]_{\texttt{ta,tv}}$? Donner ce code et vérifier qu'il permet d'accéder à la donnée en question.

4 Portées (*)

En C, les variables locales sont obligatoirement déclarées dans le bloc principal du corps de fonction (voir programme de gauche ci-dessous). On veut déclarer des variables locales dans n'importe quel sous-bloc (voir programme de droite) avec les règles usuelles de portée :

1. La *portée* d'une déclaration de variable (partie de code où la variable existe) dans un bloc se limite à ce bloc et ses sous-blocs.
2. Si une instruction est dans la portée de plusieurs déclarations utilisant le même identificateur (programme de droite, variable `x`), on considère la déclaration la plus profonde (programme de droite, variable `x` du sous-bloc).

```
int foo(int n)
{ //Bloc principal
  int[3] t;
  int i,x;
  ...
}
```

```
int foo(int n)
{ //Bloc principal
  int[3] t;
  int i,x;
  for(i=0; i<N; i++)
  { //Sous-bloc
    int[2] x; //P
    ... }}
```

Par commodité, on indice l'opérateur de traduction par l'environnement courant ρ : $[\![.]\!]_\rho$.

1. Donnez la règle de traduction pour un sous-bloc $[\![\{\text{type}_1 \text{id}_1 \,;\, ... \,;\, \text{type}_n \text{id}_n \,;$ `BODY` $\}]\!]_\rho$. On utilisera récursivement $[\![$ `BODY` $]\!]_{\rho'}$ sur un environnement ρ' à définir.
2. Dessinez l'enregistrement d'activation au point `//P` (programme de droite).

TD4 : Optimisation de code

Dans ce TD, on va voir comment simplifier le pseudo-code produit par la traduction dirigée par la syntaxe. A l'étape suivante, on pourra alors produire du code machine de bonne qualité. Plus précisément, on va voir des représentations et des techniques d'analyse de code et d'optimisation très répandues dans les compilateurs.

1 Flot de contrôle, flot de données

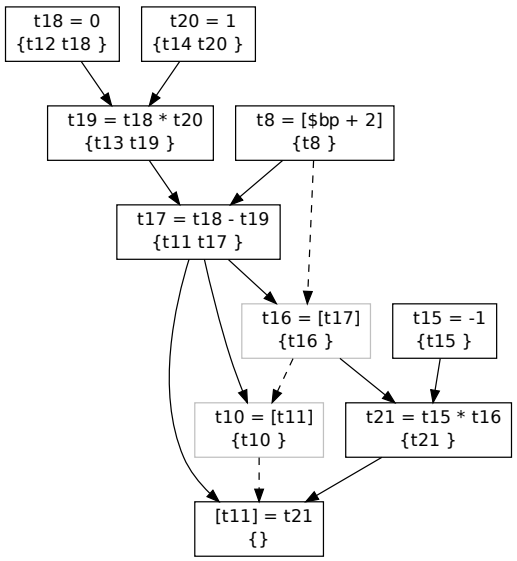
On considère le programme suivant :

```
typedef struct{int re; int im;} complex_t;

void conjuguer(complex_t z) {
  z.im = -1*z.im;
}
```

On obtient le pseudo-code et le DAG suivant :

```
conjuguer:
alloc ?
t8 = [$bp + 2]
t15 = -1
t18 = 0
t20 = 1
t19 = t18 * t20
t17 = t8 - t19
t16 = [t17]
t21 = t15 * t16
t12 = 0
t14 = 1
t13 = t12 * t14
t11 = t8 - t13
t10 = [t11]
[t11] = t21
free
ret 1
```



1. Quel temporaire contient `-1*z.im` ?
   Quel temporaire contient l'adresse à affecter (&(`z.im`)) ?
   Entourer les parties du code qui calculent `z.im`.
2. Quelles simplifications peut-on effectuer sur ce code ?
3. Quelles simplifications sont réalisées par le DAG ?
   Entourer la partie du DAG qui calcule `z.im`. Peut-on simplifier davantage ?
4. A quoi servent les flèches pointillées ? Donner une condition suffisante pour pouvoir fusionner les noeuds gris.

2 Vivacité

On considère le pseudo-code suivant :

```
c = 3
i = 0
a = t1-t2
b = t1+t2
while:
cjump i>=10 --> end_while
t4 = a * b
t3 = 1
t5 = t4 - t3
i = i + 1
jump while
end_while:
t6 = a*c
```

1. Dérouler l'analyse de vivacité
2. L'*empreinte de stockage* MAXLIVE d'un programme en pseudo-code est le nombre maximum de temporaires vivants en même temps. Donner l'empreinte de stockage de ce programme.
3. Quelle optimisation simple permet de réduire MAXLIVE ?

3 Expressions disponibles (*)

On considère le pseudo-code suivant :

```
t1 = a * b
t2 = 0
loop:
cjump t2<5 --> else
t2 = a + b
jump endif
else:
a = a + 1
t2 = a * b
endif:
t3 = a * b
cjump t2<10 --> loop
end:
```

1. Donner le graphe de flot de contrôle
2. Dérouler l'analyse des expressions disponibles.
3. Quelle optimisation peut-on appliquer ? Donner le graphe de flot de contrôle optimisé.
4. On veut écrire un algorithme Def(E,B) qui :
   - Insére les copies nécessaires pour que l'expression disponible E en provenance de B soit définie dans un temporaire w_B.
   - Retourne w_B.

(a) *Cas de base.* Donner Def(a*b,B) avec B le bloc avec `t2 = a*b`.

(b) *Récursion.* Que manque-t-il pour avoir Def(a*b,B) avec B le bloc avec `t2 = a+b` ? Pour avoir cette définition, comment faut-il appliquer Def ?

(c) Ecrire l'algorithme Def.

4 Expressions très utilisées (*)

On considère le pseudo-code suivant :

```
i = 0
a = t1 - t2
b = t1 + t2
while:
cjump i>=10 --> end_while
t4 = a * b
t5 = t4 - i
i = i + 1
jump while
end_while:
t6 = a * b
```

On dit qu'une expression E est *très utilisée* en un point d'exécution p ssi dans tout chemin d'exécution partant de p, E est utilisée avec la même valeur (les temporaires de E ne sont pas redéfinis entre p et l'utilisation de E). Par exemple, `a*b` est très utilisée juste après b = t1+t2.

Pour chaque instruction p, on note In[p] l'ensemble des expressions très utilisées *juste avant* p ; et Out[p] l'ensemble des expressions très utilisées *juste après* p.

1. Pourquoi `a*b` est très utilisée juste après `b=t1+t2` ?
2. `a*b` est-elle très utilisée juste avant `b=t1+t2` ?
3. L'analyse est-elle *forward* ou *backward* ? *may* ou *must* ?
4. Rappeler les équations de flot pour ce type d'analyse.
5. Si $p : x = E$, donner In[p] en fonction de Out[p]. En déduire les fonctions kill() et gen().
6. Dérouler votre analyse sur le programme.

TD5 : Génération de code machine

Une fois le pseudo-code généré et optimisé, il reste à produire du code machine. Ce TD montre les étapes nécessaires à la production d'un code machine efficace.

1 Sélection des instructions

On considère un bloc de base qui calcule l'expression : $E = ((3 * a) + (b - 1)) * c$ où a, b et c sont des paramètres de fonction (slots de pile).

1. Tracer l'arbre de E.
2. Trouver une sélection d'instructions optimale parmis { `add`, `sub`, `mul`, `dec`, `lea` } en appliquant l'algorithme vu en cours.
3. Donner l'arbre de calcul après sélection

2 Ordonnancement

On se propose de trouver un ordonnancement de E qui minimise le besoin en registres.

1. Quel noeud ne peut pas être traité par l'algorithme de Sethi-Ullman ?
2. Donner une règle de calcul de rg pour les opérateurs unaires : $\mathrm{rg}(op(n)) =$?.
3. Appliquer l'algorithme de Sethi-Ullman ainsi étendu. Quel est le besoin minimum en registres ?
4. Donner la séquence d'instructions obtenue (attention à l'instruction `mul`).
5. Tracer les intervalles de vie des temporaires. Vérifier que le besoin en registres est conforme aux prédictions.

3 Ordonnancement modulo A/C (*)

On considère maintenant l'expression $F = ((a * b) + (c + 3)) + d)$. On suppose une sélection *atomique* (chaque noeud est réalisé par une instruction).

1. Appliquer l'algorithme de Sethi-Ullman. Quel est le besoin minimum en registres ?
2. Transformer l'arbre de calcul pour réduire le besoin à $K = 2$ registres. Peut-on toujours réduire le besoin en registres de cette façon ?
3. Cette transformation préserve-t-elle le calcul ?

4 Allocation des registres

Le code suivant implémente l'expression $E = ((3 * a) + (b - 1)) * c$ du premier exercice :

	t1	t2	t3	t4	t5	t6	t7
mov t1,[c]							
mov t2,[b]							
dec t2							
mov t3,t2							
mov t4,[a]							
lea t5,[3*t4+t3]							
mul t1							
mov t6,t7							

1. Quels temporaires contiennent les opérandes de `mul` ? Quel temporaire contient le résultat ? Quels temporaires doivent être réalisés par `rax` ?
2. Tracer les intervalles de vie et le graphe d'interférence.
3. Donner une allocation avec $K = 3$ registres en appliquant l'algorithme de Chaitin. Quelles copies sont éliminées ? Comment fixer '`alloc?`' ?
4. Donner une allocation avec $K = 2$ registres en appliquant l'algorithme de coloriage itératif (marquer t1 et utiliser le tableau ci-dessous pour l'itération). Quelles copies sont éliminées ? Comment fixer '`alloc?`' ?

	t2	t3	t4	t5	t6	t7
mov t2,[b]						
dec t2						
mov t3,t2						
mov t4,[a]						
lea t5,[3*t4+t3]						
mul t1						
mov t6,t7						